A Case Study in Coordination Programming: Performance Evaluation of S-Net vs Concurrent Collections

Pavel Zaichenkov*§, Bert Gijsbers†‡, Clemens Grelck‡, Olga Tveretina*, Alex Shafarenko*
* *Compiler Technology and Computer Architecture Group, University of Hertfordshire, United Kingdom*
{*p.zaichenkov,o.tveretina,a.shafarenko*}*@ctca.eu*
† *Programming Languages Group, Ghent University, Belgium*
bert.gijsbers@ugent.be
‡ *Informatics Institute, University of Amsterdam, The Netherlands*
{*b.gijsbers,c.grelck*}*@uva.nl*
§ *Moscow Institute of Physics and Technology, Russia*

***Abstract*—We present a programming methodology and runtime performance case study comparing the declarative data flow coordination language S-Net with Intel's Concurrent Collections (CnC). As a coordination language S-Net achieves a near-complete separation of concerns between sequential software components implemented in a separate algorithmic language and their parallel orchestration in an asynchronous data flow streaming network.**

We investigate the merits of S-Net and CnC with the help of a relevant and non-trivial linear algebra problem: tiled Cholesky decomposition. We describe two alternative S-Net implementations of tiled Cholesky factorization and compare them with two CnC implementations, one with explicit performance tuning and one without, that have previously been used to illustrate Intel CnC. Our experiments on a 48-core machine demonstrate that S-Net manages to outperform CnC on this problem.

***Keywords*-performance measurement; coordination programming; stream processing; concurrent collections; parallel programming; language design**

I. Introduction

The main challenges in the design of concurrent programs are the correct sequencing of interactions between computational threads, and the control of shared resources. Two research directions have been pursued to address these challenges: 1) new programming models and parallel programming language abstractions; 2) specification of concurrent behavior and automatic generation of concurrent code. In the latter direction, concurrent control code is generated automatically based on a specification. Habermann's path expressions is an example of an early work in this direction [1], [2].

Yet a third way is to use a *coordination language* for the part of the program that manages concurrent *components*. The components are the building-block algorithms that the overall algorithm uses in order to produce the required results. Each component is concurrency agnostic while the overall algorithm is, in fact, a concurrent composition of the building blocks. In the coordination language one specifies the relations between components: what data is communicated between them and where synchronization takes place. A macro-dataflow coordination language, such as S-Net [3], [4], emphasizes communication between the components as it contains a complete set of wiring primitives that promote a view of an application as a *streaming network* of *asynchronous components*.

The first such attempt at coordination, albeit in a concurrency-centric, rather than communication-centric flavor, was by Gelernter and Carriero [5], who defined a concurrency/synchronization control language Linda as a set of primitives to be used with conventional imperative languages as pseudo-intrinsic functions. Further attempts brought into focus the matters of software engineering (in particular abstraction, encapsulation and inheritance) and the concept of compositionality, i.e. the ability of the concurrent glue to seamlessly integrate the components into a single program.

The coordination language S-Net takes the above issues on board. It uses streams as a glue with which it connects components into a single application. The way the components are connected depends on data types, which play a dual role in S-Net: they ensure that the correct collection of objects is received by each of the components in every act of communication, but they also help to route messages to their destinations through a very small and simple set of wiring patterns called *combinators*. Structuring the application into a hierarchical network helps to apply these simple tools systematically. The initial design is refined by progressively revealing the structure of subnetworks until the design process stops at individual components.

This paper reports on a new performance study of the language. As an example application we choose tiled Cholesky decomposition, a linear algebra algorithm that lends itself easily to parallelization for a multi-core system. It decomposes a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose. For such matrices, the algorithm is roughly twice as efficient

as the more general LU decomposition. Here we use a tiled version of the procedure originally described by Buttari et al. [6]. The choice was motivated by the fact that the algorithm is well used in computational linear algebra as well as having a sufficiently complex and varied internal structure, which would benefit from component coordination as a method of concurrent implementation. Furthermore, tiled Cholesky factorization has repeatedly been used to illustrate the merits of Intel's Concurrent Collections (CnC), and various CnC implementations have been made available. This ensures a reasonably fair performance comparison.

Contribution. We compare the performance of S-NET with that of Intel's Concurrent Collections (CnC) [7]. The choice of CnC for comparison is motivated by the fact that CnC follows a similar dataflow approach (with some control flow facilities as well) as S-NET. CnC is implemented as a C++ library (but further implementations exist), rather than a fully fledged coordination language. It is quickly gathering momentum in both industrial and academic research. S-NET, in contrast, emphasizes modular parallel program design by means of strict separation of domain-specific programming from concurrency engineering. Various S-NET applications have been built by our industrial partners, including Thales Research, SAP and Philips Healthcare [8], [9].

Our second contribution is a declarative specification of a dataflow algorithm for tiled Cholesky decomposition in S-NET. This specification avoids two acts of barrier synchronization, which occur in the imperative version of the algorithm. Our measurements show a superior performance of this declarative version over other three implementations we investigated.

II. Related Work

A coordination language describes communication between independent single-threaded computational processes. It is responsible for managing asynchronous components as well as for supporting synchronization and communication among them [5]. The coordination language is unaware of activities inside components, and thus can be added to almost any language.

A separation of concerns is an important property of coordination languages. If coordination is separated from computation, these activities can be implemented in different languages. The separation of concerns facilitates different roles for different programmers. Domain experts can concentrate on a domain-specific problem while parallel programming experts take care of concurrency aspects. The separation of concerns facilitates economy and flexibility as well. Components become more generic and thus can be reused in different contexts.

Linda is the first coordination language [10]. Communication between independent processes is performed using shared memory referred to as tuple space (elements of the memory are tuples, not bytes). Elements of tuple space are accessed by logical name. Therefore, the only information the processes share is a protocol on element tags. The separation of concerns in Linda is not complete because synchronization is located in the computational part of a program.

Kahn's model of Process Networks (KPN) introduces streams represented as sequential data channels with infinite capacity to glue independent processes into a network [11]. The KPN model is a clean coordination model because coordination does not interfere with the computations inside network vertices. S-NET can be seen a refinement of the KPN model:it takes engineering aspects, such as memory limitations into consideration. Separation of concerns in S-NET is quite clean as components are completely unaware of the context and synchronization is not interfering with computational activity.

Both S-NET and CnC use coordination glue to combine separate components together. CnC has been significantly influenced by Linda [12], whereas S-NET is based on the KPN model. We are going to show that both approaches work reasonably well to achieve excellent performance on a shared-memory parallel platform.

III. Overview of S-Net

A. The Parallel Component Technology

Decomposition and encapsulation are general software engineering principles not limited to parallel computing. Problem decomposition results in a representation of an application as a set of black-box components, whose functionality is defined in terms of the interface description, with some glue code that holds the components together in a way that ensures the expected system behavior.

The only requirement to be satisfied by an S-NET box implementation is the absence of persistent internal state. Stateful components could neither be moved or cloned in a multicore system, since the new copy would have to rely on the previous state, which is internal, and hence unavailable. However, state is somehow required in order to be able to merge messages from different channels. Thus, state may only be expressed as a dedicated language construct at the coordination level. Therefore, state in S-NET is always fully explicit.

The consequence is to structure and manage state transitions in the component world in the same way as control flow is structured and managed in ordinary programming. User-defined components become pure functions that map a tuple of parameters onto a similar collection of results.

As soon as the latter is produced, the internal state should effectively be destroyed. Such components are easy to reason about and debug, they are inherently mobile, and usable as a black box in a parallel computing environment – but there is also a price to pay. The glue environment has to provide sufficient scaffolding to support an evolving state (or local states!) of the computation. In other words, it will need to

hold the effective state of one or more components for them and present it back to the components' inputs in combination with any data to be processed.

B. *The* S-NET *Language*

The language S-NET supports coordination programming by instantiation of components as *boxes* and connecting them by anonymous data streams [4]. An S-NET application is represented as a network between the input and the output, which are two external streams connecting the whole application with its environment. In the following we briefly revisit the main concepts of the language.

The box concept. A component is instantiated as a Single-Input, Single-Output (SISO) box. The box has a limited life cycle: it accepts one item from the input stream, does some processing and yields zero or more items to the output stream, after which it destroys its internal state and waits for the next input item to arrive. Components are written in a box language, using the S-NET communication API. C and SAC [13] are currently supported as box languages.

Synchronization. In S-NET the only component which can store and combine state is the *synchrocell*. For example, the expression $[|\{r\},\{s\}|]$ synchronizes precisely two messages: one whose type contains at least a field r, another with at least s. All other messages remain untouched and are forwarded further. The semantics of the synchrocell is sequential and does not involve any data transformation, hence concurrency and mobility concerns do not apply.

The streaming data concept. All boxes accept records as units of their input. A record in S-NET is a set of *fields* and *tags*. Both fields and tags have names and values. Field values cannot be examined in S-NET: they are references to data which are private to the box language. Tags are standardized as integers and their values are available in both the box language and the S-NET language. Records are non-recursive in the sense that it is not possible to define an unlimited linked structure, such as a list.

Every user-defined component contains a program unit (a function or similar) written in a box language, and a type signature written in S-NET that defines the type of records (in terms of their field/tag name sets) that the box accepts and, in a similar way, the types of any output records that may be produced. Streams between boxes are sequences of records. Even though all boxes are SISO, the data relationships between them are not one-to-one, since streams can be split and merged using combinators.

Combinators. These are second-order functions that ensure compositionality of SISO networks. First of all there are serial and parallel combinators, $A..B$ and $A|B$, respectively. The serial combinator ".." (Figure 1(a)) connects the output of operand A to the input of operand B, with the input of A and the output of B becoming those of the resulting network. The parallel combinator "|" combines its operands

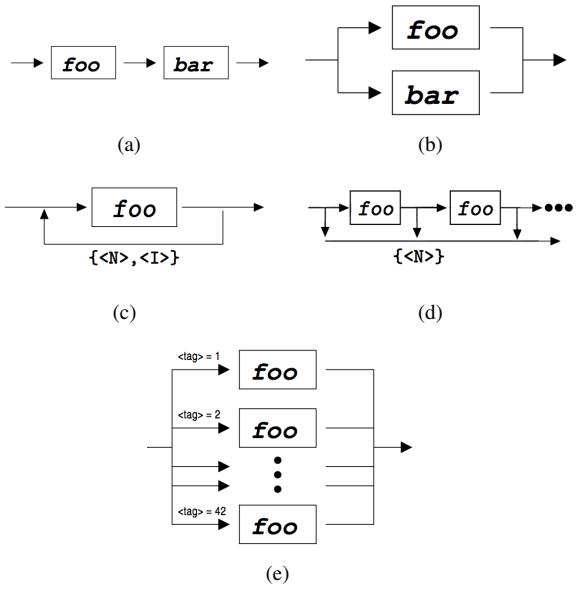

Figure 1. Illustration of network combinators used in Cholesky decomposition implementation for S-NET: serial combinator (serial composition) (a), parallel combinator (parallel composition) (b), feedback combinator (feedback loop) (c), star combinator (dynamic replication) (d) and split combinator (parallel replication) (e).

in parallel, see Figure 1(b). Incoming records are sent to the operand network that best matches its type [3]. Type specifications for complex networks are inferred automatically by the S-NET compiler.

The feedback combinator examines the output records of an operand network and redirects those records that match a pattern back into the input of that network. For instance, $C\backslash z$ creates a feedback loop around operand network C for records for as long as their type matches type pattern z. This allows a single operand network C to repeatedly process a record until it is finally converted into something else, see Figure 1(c).

There exist combinators for dynamic replication of a SISO network. The expression $A*x$ will serially replicate the operand network A an unspecified number of times (Figure 1(d)). Only records whose type matches the given type pattern x escape this network. Thus, the expression is equivalent to an infinite serial expansion $A..A..A..$ for records as long as they do *not* match exit pattern x. Typically, at some point in time, due to processing by network A, they *do* match the exit condition and will then appear on the output stream of the expression.

Similarly, the expression $B!<y>$ replicates the SISO network B an unspecified number of times in parallel (Figure 1(e)). For every unique tag value y one parallel branch of B is created. This can be seen as an infinite parallel expansion $B_{y_0}|B_{y_1}|B_{y_2}|\ldots$ The branches persist. Namely, records with the same tag value y take the same

```
1: for k = 0, ..., p − 1 do
2:     InitialFactorization(A_kk, L_kk)
3:     for all j ∈ (k + 1, ..., p − 1) do
4:         TriangularSolve(L_kk, A_jk, L_jk)
5:     end for
6:     for all j ∈ (k + 1, ..., p − 1) do
7:         for all i ∈ (k + 1, ..., i) do
8:             SymmetricRankUpdate(L_jk, L_ik, A_ij)
9:         end for
10:    end for
11: end for
```

Figure 2. Tiled Cholesky decomposition algorithm.

branch. All records which encounter this expression are required to have a type which contains tag $<y>$. This is checked at compile time.

Stateful computations can be modelled in S-NET with an expression like $([|\{r\},\{s\}|]*\{r,s\}..MyBox)\backslash\{s\}$, where a single state $\{s\}$ is first combined with an incoming record $\{r\}$. Processing by component $MyBox$ may generate any number and type of output messages, but at least one evolved state $\{s\}$. The feedback combinator "\" only redirects the state back into the network where the process repeats itself with the next incoming record $\{r\}$. For details regarding stateful streaming networks in S-NET we refer the interested reader to [14].

IV. Case Study: Cholesky Decomposition

A. The Algorithm

Cholesky factorization computes a solution to the following problem: given a symmetric positive definite matrix A, find a lower-triangular matrix L, such that $A = LL^T$. We use the tiled version of the Cholesky decomposition algorithm described by Buttari et al. [6].

Initially, the input matrix A is decomposed into $p \cdot p$ blocks A_{ij} of size $b \times b$ each. Then, we solve the Cholesky decomposition problem for all blocks from submatrix A_{i0} separately. Next, we recompute all element values in the submatrix A_{ij} (where $i \in (1, p-1), j \in (1, i)$) and run the algorithm recursively on the submatrix.

An overall algorithm is given in Figure 2. The computational process is divided into three steps:

Initial Factorization. A scalar Cholesky decomposition algorithm is used to solve $A_{kk} = L_{kk}L_{kk}^T$ equation on this stage. The result of computation is a lower-triangular matrix tile L_{kk}.

Triangular Solve. During this phase we apply the result of the previous step's computation to solve the equation $A_{jk} = L_{jk}L_{kk}^T$. The result is a matrix tile L_{jk}. This step can be performed for all tiles in the same column concurrently.

Symmetric Rank Update. This step is used to update values of tiles A_{ij}, where j ranges from $k+1$ to $p-1$ and i from $k+1$ to j. This is done using the following formula: $A'_{ij} = A_{ij} - L_{ik}L_{jk}^T$. Similar to the previous step, this can be done concurrently for all i and j.

This numerical problem thus boils down to three building blocks. Our task in coordination programming is to glue them together into one coordination program.

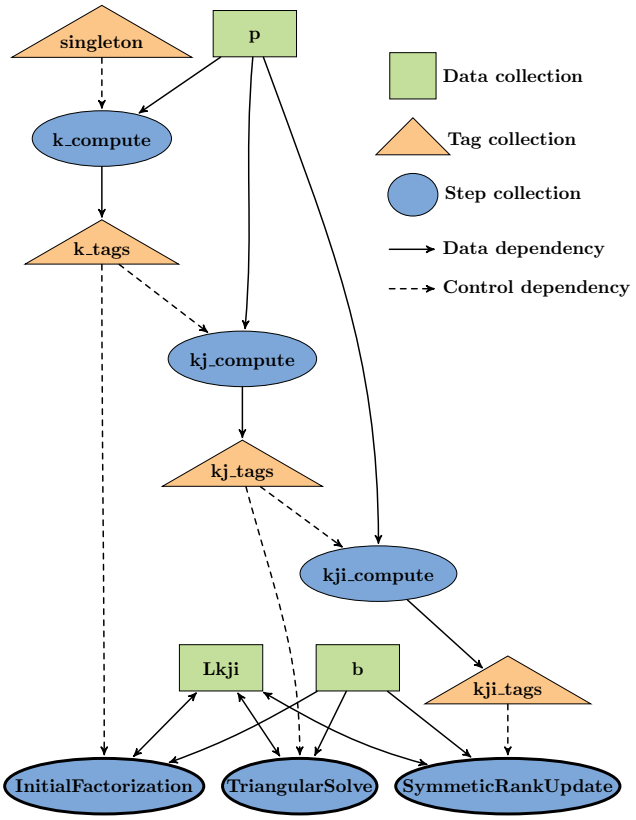

Figure 3. A computational graph of the CnC implementation of the Cholesky decomposition algorithm

B. Implementation with Intel Concurrent Collections

A detailed description of the CnC implementation of the Cholesky decomposition algorithm is found in [15]. Here we present only a brief description of the CnC concepts relevant to our implementation.

The CnC model has the following important feature: it decouples the specification of a computation from the expression of its parallelism. Consequently a domain expert determines the design of the algorithm, and a tuning expert can be called upon to deal with parallelism, communication, scheduling and distribution issues, not dissimilar from S-NET.

The domain expert specifies the computation in graph form as depicted in Figure 3. The graph contains the following types of nodes.

- A computational *step*. It is a basic unit of execution specified explicitly by the domain expert. In Figure 3 ellipses represent *step collections*, which are static

declarations of sets of dynamic instances.

- A data *item*. *Item collections* are used to represent data. Items are elementary units of storage, communication, and synchronization. In Figure 3 there are three item collections shown as rectangles: $Lkji$ stores both input matrix A and output matrix L, b stores a block size and p is used to calculate the total number of blocks in the initial matrix (which equals $p \times p$).
- A control *tag*. Each instance of a step or item has a unique tag, which is a tuple of *tag components*. Tags indicate *whether* a step will execute, but not *when* it executes. A step may produce tags as well. A step collection is associated with exactly one tag collection.

The relations between steps and items are shown by directed solid edges in Figure 3. They have the following meaning: The line "item $\rightarrow$ step" indicates that the step consumes the item and the line "step $\rightarrow$ item" means that the step produces the item.

The control relation between a tag collection and a step collection is shown by directed dashed edges. A step collection is associated with exactly one tag collection; multiple step collections may be *prescribed* by the same tag collection.

Figure 3 contains tags and items that do not have inbound edges. This means that they are taken from the *environment*, which is the external code that invokes the computation. For our example the environment provides $Lkji$, b and p item collections and the `singleton` tag collection.

There are three main algorithms given as separate steps in the implementation: `InitialFactorization`, `TriangularSolve` and `SymmetricRankUpdate` written by the domain expert. Their behavior is defined by data collections $Lkji$, b and p they receive as an input data. Tag collections `k_tags`, `kj_tags` and `kji_tags` control the behavior of each step collection. All of these steps produce the result of the computation and put it back into the data collection $Lkji$.

Semantics and execution of CnC are defined as follows.

- The item or tag is said to be *available* if it was produced by a step.
- The step is said to be *prescribed* if it was prescribed by a tag collection and a particular tag became *available*.
- The step becomes *inputs-available* if all items for this step are available.
- The step is *enabled* and may execute if it is both inputs-available and prescribed.
- The program terminates when no step is executing and no unexecuted step is enabled. The termination is valid if all prescribed steps have been executed.

Nevertheless, the relation between step, data and tag collections is defined statically, there is no way to perform scheduling and resource management dynamically. In order to solve this shortcoming CnC offers a tuning mechanism in the form of special annotations for compiler and runtime system.

In order to evaluate the effect of tuning, we evaluate two alternative CnC implementations of tiled Cholesky factorization. The first one does not rely on the tuning, whereas the second one uses dependency functions in order to improve scheduling. Dependency functions map control tags into data collection indices in order to improve scheduling and eliminate stalls during run-time. Without this information it is impossible to determine which elements are going to be accessed by steps, therefore steps are forced to stall.

C. Implementation with S-NET

Separation of concerns is also a key feature in S-NET. S-NET describes the coordination behavior of networks of asynchronous components and their orderly interconnection via typed streams. The component implementation is done using an external *box language* and S-NET is not bound to any specific one.

S-NET uses a message-driven communication model. The domain expert only specifies the input and output types of boxes. This is the type signature of a box. The type signature is basically a declaration of how the type of the input message is mapped onto the types of the output messages. On each computational step a box receives only a single message as input, yet the number of output messages is not limited. A box is stateless and runs asynchronously with other boxes. Boxes do not carry global state and the *purity* of a function inside a box is a requirement. A box computation may start as soon as a message from the input stream is received.

We developed two different implementations for Cholesky decomposition in S-NET. The first one strictly adheres to the algorithm shown in Figure 2, and thus implements three consecutive steps. The drawback of this approach is that in each iteration two barrier synchronizations take place.

Our second implementation is free from barrier synchronization and, hence, exposes a higher degree of concurrency. Here, all computations are completely data-driven as in a dataflow approach: any computational step in the program is able to execute as soon as the required input data becomes available. We now cover both implementations in more detail.

1) Implementation with barrier synchronization: Compared to CnC, a coordination network in S-NET does not specify a control flow. Box computations depend only on the availability of input data. Data relations are completely defined statically by means of type specifications. In contrast to the CnC graph, the S-NET network is hierarchically structured. Any network can structurally play the role of a box in a higher-level network.

In Figure 4 we illustrate the S-NET network for the first Cholesky decomposition implementation. Each box shows its type signature, with one input type and one or more

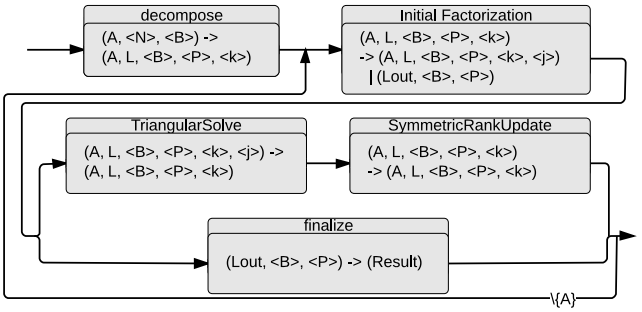

Figure 4. The S-NET network for tiled Cholesky decomposition with barrier synchronization.

output types. *Tags* are distinguished from *fields* by angular brackets. The coordination layer is able to access tags and route messages based on (integer) tag values.

The `decompose` box receives a message with the input matrix A, its size N and the block size B from the environment. It reallocates the array where the input matrix is stored and permutes the matrix elements there in order to improve spatial and time locality. As the result, box `decompose` outputs the input matrix with permuted elements A, the output matrix L filled with zeros (on each iteration of feedback loop combinator we add new values to the matrix), the block size B, the number of blocks P and an additional iteration index k with initial value zero.

Next, we perform recursive computations (the outer loop in Figure 2). The recursion is expressed using a feedback loop combinator. The combinator redirects the output of the `SymmetricRankUpdate` box to the input of the `InitialFactorization` box as long as the message containing a field of type `A` is produced. Execution terminates once `SymmetricRankUpdate` stops producing new messages. The result of computation is stored in a message that is produced by `finalize` box and is sent to the output stream.

We compare the loop index k with the number P in order to determine whether all the blocks have been computed. Depending on the result, messages of different types are sent. If k is still less than P, we add computed elements to the matrix L and supply the input record with the additional index j. `TriangularSolve` and `SymmetricRankUpdate` boxes perform the computations of the corresponding stages. Lastly, the `finalize` box completes the computation by converting the result matrix to a format suitable for output and performing memory deallocation.

2) Data-driven implementation: In our second S-NET implementation of Cholesky decomposition the three central steps from the algorithm are all run in parallel, see Figure 5.

Box `start` receives the same input, consisting of matrix A, matrix size N and the block size B. It produces one record containing the matrix $Result$ which is filled with zeros, together with a tag X which denotes the

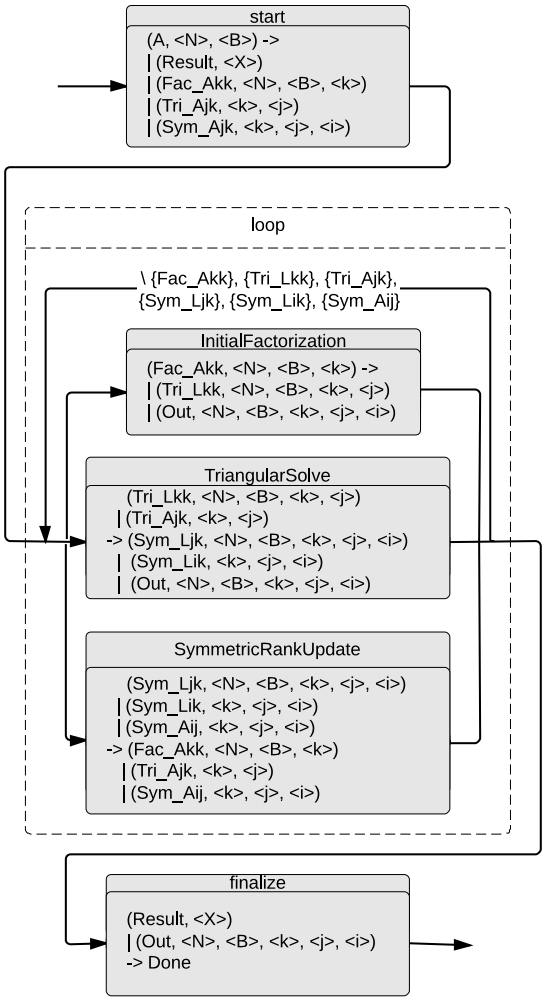

Figure 5. The S-NET network for tiled Cholesky decomposition following a purely data-driven approach.

number of Out tiles which need to be merged with the result matrix in order to construct the final output matrix. This $Result$ message is immediately forwarded to the `finalize` box. In addition, the `start` box produces initial messages containing separate tiles each of which corresponds to some stage in the algorithm. That is, it produces messages with diagonal tiles with type `Fac_Akk` which are accepted by `InitialFactorization`; messages with type `Tri_Ajk` and tag j which are accepted by `TriangularSolve`; and messages with type `Sym_Aij` and tags j and i which are accepted by `SymmetricRankUpdate`. Tag j denotes a column index and tag i a row index. In addition each message contains a loop iteration tag k in order to distinguish different tiles from different iterations. All individual box components are replicated in parallel by the "!" combinator with index values taken from the i, j and k tags. This ensures that no messages are queued up in streams waiting for preceding messages to

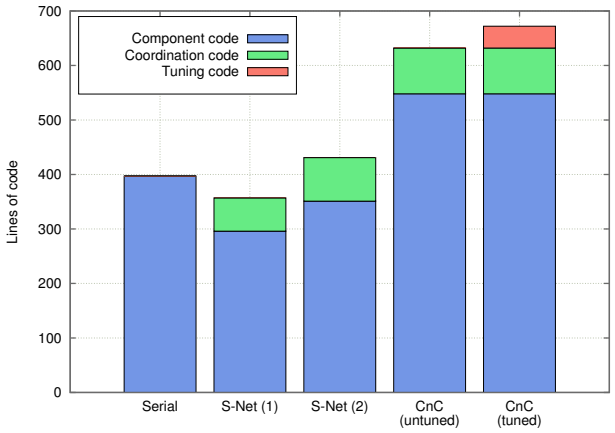

Figure 6. Number of lines of code for each implementation.

be processed. Input dependencies can therefore quickly be resolved and computations can start as soon the input data becomes available.

Messages are transferred between the three central components until result tiles with type Out are produced which are accumulated by the `finalize` box. To this end the `finalize` is paired with a repeated synchrocell and both are enclosed in a dedicated feedback loop which is not shown in figure 5. Components `TriangularSolve` and `SymmetricRankUpdate` also contain a synchrocell inside. The synchrocell in the first component awaits for two messages with `Tri_Ajk` and `Tri_Lkk`, representing tiles A_{jk} and L_{kk} respectively. Similarly, synchrocells in the second component awaits for three tiles L_{jk}, L_{ik} and A_{ij} which are packed into `Sym_Ljk`, `Sym_Lik` and `Sym_Aij` respectively. Once the dependencies are satisfied, a box computation starts, which results in one or more output messages.

From this description it becomes clear that synchronization is only needed to satisfy specific input dependencies in preparation for individual activations of box computations. Collective thread synchronizations are completely avoided. Messages are transferred independently and computations are free to start as soon as all input dependencies are satisfied.

3) Discussion: In terms of expressiveness, all CnC and S-NET implementations are well-structured and there is a reasonably low amount of extra code required to pay for coordination. In most cases, the implementation consists of two parts: the high-level structure of the program represented as a graph and a set of functions defining box/step behavior that use an API for interacting with S-NET/CnC respectively. In addition, tuning code may be present in the CnC program.

We demonstrate the programmability of each approach by showing the number of lines of code for each program in Figure 6. It gives a rough idea of the amount of coordination and tuning overhead. Overall, the amount of coordination code is relatively small in either case. CnC components are implemented in C++ and representing each component as a separate class requires some additional overhead.

To summarize, CnC and S-NET are both systems for coordination of components. Components in CnC are linked by data and control relations. The execution strategy and computation order are determined mainly dynamically. S-NET uses a message-driven strategy. Here components are linked by typed data relations only. Additionally S-NET offers a hierarchical structuring mechanism and a compile time analysis which supports this structure; most of that analysis can be done statically.

V. PERFORMANCE MEASUREMENTS

Cholesky decomposition is a significant example of a computational linear algebra problem. It is affected by both locality (tile size) and parallelism (the number of cores used). The amount of work and available parallelism varies during the run time, which should reveal the differences in both systems' abilities to manage the resources of a concurrent platform.

The CnC model permits many run-time system designs, including those for distributed memory systems using MPI as well as shared memory versions. We use Intel CnC 0.9, which uses Intel Thread Building Blocks (TBB) as a threading layer. Step components are implemented in C++ and compiled into libraries using GNU GCC 4.6.3.

Boxes for S-NET were implemented in C and were compiled with GNU GCC 4.6.3. All measurements were done with the FRONT runtime system [16]–[18]. This is a novel runtime system, which combines a very low overhead of S-NET box/network instantiation with efficient transportation of records throughout the network and box replication. Experiments show that this runtime system scales well to very large S-NET networks with millions of concurrent records.

All measurements are obtained on a machine with four twelve-core AMD Opteron™ 6174 Processors and 256 GB RAM (see Table I for technical details). When measuring speedup for increasing core counts, we first employ neighboring cores in the same processor before going beyond processor boundaries.

The serial implementation was used as a reference to measure the speedup. This version implements the tiled Cholesky decomposition algorithm for a single core without use of optimized kernels. It is similar to the implementation in CnC, but without any coordination overhead. The program consists of three essential functions, each representing a single step of the algorithm shown in Figure 2. As the algorithm states, these functions are called in a loop with different parameters. During each iteration a global array of tiles is accessed to read or to write tiles. This is similar to how the data collection is accessed in CnC.

Table I
EVALUATION PLATFORM FOR EXPERIMENTS

Vendor	AMD
Processor Model	Opteron 6174
Processor Name	Magny-Cours
Clock (GHz)	2.2
# Sockets	4
Cores(Threads)/Socket	12(12)
L1 Data Cache	64 KB/core
L2 (Data and Instruction) Cache	512 KB/core
Shared L3 Cache	12 MB
DRAM Capacity	256 GB

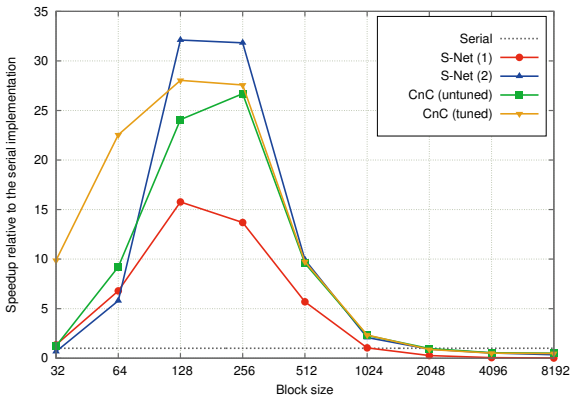

Figure 7. The speedup relative to the serial implementation of Cholesky decomposition CnC and S-NET applications on 48-core machine for an input matrix of size $N = 8192$.

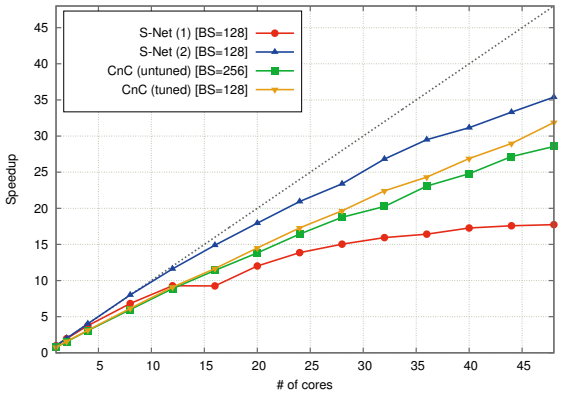

Figure 8. The speedup of Cholesky decomposition CnC and S-NET applications for different number of cores (for matrix of size 8192×8192 and optimal block size).

We provide evaluation results for the two CnC and the two S-NET implementations of tiled Cholesky factorization as described in Section IV. Figure 7 shows the speedup relative to the serial implementation executed with optimal block size $b = 128$ as a parameter. We systematically vary block sizes for both CnC and S-NET implementations. In this figure the overhead can be observed for large blocks, where the speedup of concurrent versions drops significantly. Four peaks in the figure demonstrate optimal values for block size. The maximum speedup is 32 that was achieved by S-NET data-driven implementation.

The use of dependency functions in the tuned CnC implementation brings significant improvement compared to the untuned version of the program, especially when there is a high amount of concurrency (for Cholesky decomposition it is the case where the size of subproblems is small). In the best case it brings almost a factor of 8 improvement for the current application. For the best performance range of tile size improvement caused by dependency functions is about 15%. On the other hand, an overhead introduced by the dependency functions for cases with a small amount of concurrency caused a 7% loss in performance.

Figure 8 shows speedups of S-NET and CnC for increasing numbers of cores. As pointed out before, we first use all cores of one processor before proceeding to the next processor when increasing the effective core count to exploit the memory hierarchy. All applications were executed with optimal block sizes. Within the field of tested implementations the data-driven S-NET version of tiled Cholesky factorization achieves the best performance and scales up to the maximum of 48 cores. Both CnC implementations are clearly behind, although they likewise demonstrate excellent scalability. Making use of the tuning facilities of CnC results in a rather marginal performance advantage in this experiment. As expected, the S-NET implementation involving barrier synchronization suffers from lesser scalability and increasing overhead as the number of cores used grows. We can, in particular, observe this effect when moving from 12 cores to 16 cores, i.e. beyond a single processor.

A thorough performance evaluation of the tiled Cholesky decomposition in CnC can be found in [15]. The results of the measurements illustrate that S-NET on this example has a performance similar to CnC and that coordination programming model is an effective instrument for implementing applications for multi-core platforms.

The second S-NET implementation clearly benefits from the dataflow model which S-NET provides. The execution of a component is enabled by the arrival of input data. This paradigm allows for the specification of highly-concurrent applications. In contrast, execution of CnC component is prescribed by tag without awareness of data availability. This may introduce stalls during execution. A significant speedup in CnC was achieved by introducing dependency functions, which map tag indices to element indices in data collection.

VI. CONCLUSIONS

We presented a performance case study on a popular linear algebra problem using coordination programming as a method of code development. We compared two design styles of coordination programming and their runtime performance on a large multicore server: our coordination

language S-NET vs Intel's coordination library/specification tool CnC (Concurrent Collections).

We observe that a static network topology and data relations facilitates S-NET compilation and run-time scheduling and communication. S-NET does not use control flow, allowing components to be triggered merely by the availability of their input data. Despite the lack of tag collections that determine the sequencing of processing steps, pure dataflow works quite well, outperforming CnC in the best performance range of problem sizes. S-NET supports a clean separation of concerns between coordination and computation: only individual objects required by a computational component are delivered to it by a coordinator. By contrast, in CnC components must be aware of the whole data collections they wish to access.

Tuning is a feature of CnC that is clearly separated from application design. By introducing "depends" functions to the application we demonstrated the improvement this can bring. For the current application it delivers an 8 fold improvement in the best case.

Both of CnC and S-NET are designed to maximize programmability and usability of various many-core platforms. We compared the two coordination models with a serial implementation. We managed to achieve optimal utilization of the resources without platform-specific tuning and optimization. The data-driven implementation is S-NET is based on precisely the same sequence of algorithmic steps as the CnC one (though implementation with a barrier synchronization is different). In order to increase the performance for CnC, one should consider other features of the tuning mechanism (i.e. priorities) that may improve scheduling and memory management at the run-time.

ACKNOWLEDGMENT

This work has made use of the University of Hertfordshire Science and Technology Research Institute high-performance computing facility. The authors further wish to thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] S. Andler, "Predicate path expressions," in *Proceedings of the 6th ACM Symposium of Principles of Programming Languages (POPL 1979)*, 1979, pp. 216–236.

[2] C. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM*, vol. 17, no. 10, pp. 549–557, 1974.

[3] C. Grelck, S.-B. Scholz, and A. Shafarenko, "A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components," *Parallel Processing Letters*, vol. 18, no. 2, pp. 221–237, 2008.

[4] C. Grelck, S. Scholz, and A. Shafarenko, "Asynchronous stream processing with S-Net," *International Journal of Parallel Programming*, vol. 38, no. 1, pp. 38–67, 2010.

[5] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communications of the ACM*, vol. 35, no. 2, pp. 96–107, Feb. 1992.

[6] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38–53, Jan. 2009.

[7] Z. Budimlic, A. Chandramowlishwaran, K. Knobe, G. Lowney, V. Sarkar, and L. Treggiari, "Multi-core implementations of the concurrent collections programming model," in *The 14th Workshop on Compilers for Parallel Computing*, 2009.

[8] F. Penczek, S. Herhut, C. Grelck, S.-B. Scholz, A. Shafarenko, R. Barriére, and E. Lenormand, "Parallel signal processing with S-Net," *Procedia Computer Science*, vol. 1, no. 1, pp. 2079–2088, 2010, iCCS 2010.

[9] C. Grelck, S.-B. Scholz, and A. Shafarenko, "Coordinating data parallel SAC programs with S-Net," in *IEEE Trans. Parallel Distrib. Syst.* IEEE Computer Society Press, Los Alamitos, California, USA, 2007.

[10] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and friends," *Computer*, vol. 19, no. 8, pp. 26–34, 1986.

[11] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing 74, Proc. IFIP Congress 74. August 5-10, Stockholm, Sweden*, L. Rosenfeld, Ed. North-Holland, 1974, pp. 471–475.

[12] M. G. Burke, K. Knobe, R. Newton, and V. Sarkar, "The concurrent collections programming model," Rice University, Tech. Rep. TR 10-12, Dec. 2010.

[13] C. Grelck and S.-B. Scholz, "SAC: A functional array language for efficient multithreaded execution," *International Journal of Parallel Programming*, vol. 34, no. 4, pp. 383–427, 2006.

[14] C. Grelck, "The essence of synchronisation in asynchronous data flow," in *25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11), Anchorage, USA, Workshop Proceedings*. IEEE Computer Society Press, 2011, pp. 1159–1167.

[15] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance evaluation of concurrent collections on high-performance multicore computing systems," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, Apr. 2010, pp. 1–12.

[16] B. Gijsbers, "An efficient scalable work-stealing runtime system for the S-Net coordination language," Master's thesis, University of Amsterdam, Amsterdam, Netherlands, 2013.

[17] B. Gijsbers and C. Grelck, "An efficient scalable runtime system for macro data flow processing using S-Net," in *6th International Symposium of High-Level Parallel Programming and Applications (HLPP'13), Paris, France*, G. Hains and Y. Khmelevsky, Eds. Université Paris-Est, 2013.

[18] B. Gijsbers and C. Grelck, "An efficient scalable runtime system for macro data flow processing using S-Net," in *International Journal of Parallel Programming*, 2014.